BACHELOR INFORMATICA

UNIVERSITEIT VAN AMSTERDAM

Extensible Simulation of Functionally Transparent SystemC-based Computer System Architectures

R.A.J. Wacanno

July 28, 2021

Supervisor(s):
drs. T.R. Walsta,
drs. A. van Inge,
ing. E.H. Steffens

Signed: Signees

Abstract

Contents

CHAPTER 1

Introduction

One of the many advantages of software design—when compared to traditional hardware design, is the relative low bar of entry. Anyone with a computer, some time and the right amount of motivation, is able to start working on projects or teach him or herself the ins and outs of various programming languages or theories in programming. This used to be in stark contrast to hardware design. Trying to design custom hardware components or learning about the inner workings of certain components was done in either a strictly theoretical fashion, or in a more involved practical fashion. The first of these tactics typically entails reading up on the needed technical specifications from manuals and applying this to an abstract implementation in the form of diagrams. This way, testing of the functional correctness of a design had to be tested purely based on deduction from this theoretical implementation, without real world testing. Practical testing could also be a possibility, but this came with other disadvantages, like having to buy necessary components, equipment, and needing to have the required technical knowledge to handle said components and equipment.

With the dawn of more capable computational hardware, it becomes more and more feasible to design hardware in a more software-like fashion and test its functionality in a purely software based simulation framework. Applying these techniques to designing and understanding hardware eliminates the need for buying actual hardware, which, like software design, dramatically lowers the bar of entry.

Apart from this, hardware simulation introduces more possible advantages when compared to using real components. When running more complex components, like integrated circuits, the inner workings are hidden in a black box-like fashion. Without the use of specialised probing equipment, it is impossible to see what goes on inside such parts. With further increased complexity, having a look inside the a used components might even become impossible. By using simulated hardware,

This research in particular focuses on the simulation of computer system architectures. Central to these architecture is of course the central processing unit of CPU, which is inherently a complex components comprised of a number of discrete components brought together on a single piece of silicon. In order to perform the simulation of such a component, an appropriate description language is needed. In this case, the C++ library SystemC is chosen for the software-based implementation of the simulated system architectures.

In the end, one might still want to test a hardware design written in software in the real world. Therefore, this project aims to implement a systematic way in which the SystemC-based hardware implementation can be translated to a HDL-based implementation. Such an implementations should in turn be executable of the right FPGA hardware, and, as such, the software-described architecture implementation can be tested in a true hardware context.

1.1 Research question

In order to fulfil the needs presented in the introduction, a research question with a set of sub-questions has been formulated:

- How can a system architecture be simulated in a functionally transparent way, using a SystemC-based hardware description.
  - What systematic approach can be used to visualise the internal structure of a SystemC-based archtitecture in terms of components and data streams?
  - What communication mechanisms are required to ensure external extensibility of a simulation framework?
  - What constraints are placed on a SystemC hardware description when this description is to be converted to a VHDL-based description?

1.2 Outline

The following writing will introduce the theoretical concepts needed in order to perform this research in chapter 2. Based on this theoretical knowledge, the desired framework is implemented in an iterative fashion. The process of this implementation and the motivations behind specific implementation decisions is described in chapter 3. In chapter 4, results of tests done with the implemented framework are presented. Finally, chapter 5 features concluding remarks, a reflection on the research process of this project and possible future work which might follow from this project.

CHAPTER 2

Theoretical background

The following sections give a brief outline of the theoretical knowledge used to perform this research.

2.1 Existing simulation solutions

This project is not the first in the field of architecture simulation, and, as such, similar solutions have been devised in order to satisfy the need for a system architecture simulation environment. The following are a few of these existing packages, their aims and a comparisons of these aims and the aims of this research.

2.1.1 ArchC

While not necessarily a simulation framework on its own, ArchC calls itself an Architecture Description Language (ADL) which intends to formulate a language, based on generic SystemC, which can be used to implement system architectures [Rigo et al., 2004]. ArchC acts as an extension on the basic SystemC library, providing a higher level of abstraction when implementing system architectures.

Using ArchC, the lay-out of a system architecture can be constructed using predefined functional components offered by the language. In addition to this, the instruction-set architecture supported by the hardware can be defined using specific syntax. These two parts, the so called Architecture Resources and Instruction-set Architectures, form the basis for simulation wheb using ArchC.

Listing 2.1 shows the Architecture Resource description of a simple MIPS pipelined architecture. The description defines two caches, a register file and the stages of the execution pipeline.

```
AC_ARCH(mips){
    ac_cache icache("dm", 16, 4, "wt", "war");
    ac_cache dcache("2w", 32, 4, "lru", "wb", "wal");

    ac_regbank RF:32;
    ac_pipe pipe = {IF, ID, EX, MEM, WB};
    ac_wordsize 32;

    ac_format RegFmt1 = "%npc:32";
    ac_format RegFmt2 = "%rd:5 %rs:32 %rt:32 %imm:32";

    ac_reg<RegFmt1> IF_ID;
    ac_reg<RegFmt2> ID_EX;
    ...
    ARCH_CTOR(mips){
        set_endian("big");
    }
};
```

Listing 2.1: ArchC Architecture Resource definition for a simple MIPS architecture

Listing 2.2 in turn shows the description of the Instruction-set Architecture portion of the MIPS implementation. It formalises two instruction formats `R` and `I` and defines two instructions which will be supported by this implementation (`add` and `load`).

```
AC_ISA(mips){
    ac_format Type_R="%op1:6 %rs:5 %rt:5 %rd:5 0x00:5 %op2:6";

    ac_format Type_I="%op1:6 %rs:5 %rt:5 %imm:16";

    ac_instr<Type_R> add;
    ac_instr<Type_I> load;

    ISA_CTOR(mips){
        add.set_asm("add %rs, %rt, %rd");
        add.set_decoder(op1=0x00, op2=0x20);
        load.set_asm("lw %rt, %imm(%rs)");
        load.set_decoder(op1=0x23);
    };
};
```

Listing 2.2: ArchC Instruction-set Architecture definition for a simple MIPS architecture

This example shows that using the ArchC ADL, a SystemC-based architecture can be implemented using a higher level of abstraction when compared to implementation using bare SystemC. While this eases the development process this higher abstraction level is also the main weakness when trying to use it's functionality for answering this projects research question. To facilitate this ease of use, it is heavily reliant on the predefined components present in the ArchC specification. This means that, when using the ADL, components can't be easily altered or replaced by self-implemented alternatives. The predefined components are also not transparent in their functionallity, and as a result could not be easily fit into a architecture monitoring framework as would be required for this research.

[iets over aansturing]

All this considered, while ArchC itself might not be applicable to the problem phrased in this writing's introduction, lessons can be learned from the way ArchC simplifies the hardware implementation process.

2.1.2 gSysC

The issue of internal state monitoring and control has been addressed in previous research [Eibl et al., 2005]. The gSysC framework extends basic SystemC functionality with a graphical user interface that shows the components present in the hardware description and the data streams between these components. Using a set of macros provided by the gSysC header, the user can register specific modules and ports to allow for monitoring and control in the GUI.

```
#include "gsysc.h"

int sc_main(int argc, char* argv[])
{
    sc_clock cpu_clk("CPU-Clock");
    sc_signal<sc_bv<32> > addr_sig;
    sc_signal<bool> we_sig;

    cache_connect* c;

    REG_MODULE(c, "Cache-Connect", NULL);
    REG_MODULE(c->ctrl,"CController",c);
    REG_MODULE(c->memory,"CMemory",c);

    sc_signal<bool> bus_clk_sig;

    bus_bus b("bus");
    b.m_dt(or_mb_dt);
    b.clk(bus_clk_sig);

```

```
21      REG_MODULE(&b, "Bus", NULL);
22      REG_INOUT_PORT(&b.m_dt,&b,&or_mb_dt);
23
24      c->cpu_clk(cpu_clk);
25      c->bus_clk(bus_clk_sig);
26
27      REG_IN_PORT(&c->cpu_clk,c,&cpu_clk);
28      REG_IN_PORT(&c->bus_clk,c,&bus_clk_sig);
29
30      bus_master_or mor("master_or");
31      REG_MODULE(&mor, "Bus-Master-OR", NULL);
32
33      return 0;
34  }
```

Listing 2.3: Example SystemC program using gSysC bindings

While this library satisfies the need for external monitoring and control of a SystemC hardware component, its method of achieving this is not compatible with our research goal. First of all, with gSysC, the GUI is an intrinsic part of the SystemC simulation process. This means that changes in the SystemC hardware implementation will also result in recompilation of the gSysC GUI.

The bindings offered by gSysC are also rather limited and in their current form not easily extendable. When thinking about the functional transparency aspect of this research, manipulation of the internal state of components—like changing the contents of register files and memory modules—are a requirement. gSysC does not offer simulation control to this extent.

2.1.3 PTLsim

PTLsim is a cycle accurate simulator with a specific focus on the x86-64 system architecture [Yourst, 2007]. The focus of the PTLsim simulator is not on experimentation with and design of self-implemented of system architectures or existing reference implementations. Instead, the primary purpose of PTLsim is on allowing experimentation with the x86 architecture, including a full implementation of the x86-64 instruction-set architecture.

With such a narrow architectural focus, PTLsim is able to fine-tune the cycle-accurate simulation of the x86-64 architecture. Because of this, the simulation model employed in PLTsim is able to achieve results comparable to real x86-64 silicon implementations with respect to accuracy [Yourst, 2007, table 1, p. 8].

Without any architectural modularity, PLTsim cannot satisfy our research goals, but does form an interesting example of what accurate system architecture hardware simulation can be capable of.

2.1.4 gem5

When compared to the former simulation solution, gem5 is a more general purpose simulation framework [Binkert et al., 2011]. As opposed to only allowing simulation of the x86 architecture, gem5 allows for the simulation of multiple system architectures in a modular fashion.

Another feature of the gem5 framework is the extendability with external processes

With particular interest to this research is the fact that work has been done incorporating the functionality of SystemC-based architectures in the gem5 framework, as proposed in Menard et al. [2017]. This article outlines a method of extending the basic functionality of gem5 with support for interoperability with SystemC-based architecture implementations.

2.1.5 SIM-PL

A project which close to this project in terms of functionality

2.2 User base

Having explored the features of comparable solutions in the field of computer architecture simulation in the previous sections, the question that arises is how this project aims to differentiate and pose a solution to problems that have not been satisfied by these existing software packages.

2.3 Simulation models

Since this research's main focus is on simulating system architectures, it is important to identify various methods of simulation. As is true for many thing, simulation can be done in numerous ways, and each method can be seen as more or less applicable in certain specific use cases.

2.3.1 Continuous-time simulation

The first simulation technique covered in this theoretical introduction is known as continuous simulation. This technique makes use of continuous models in order to simulate a given system. Such a continuous model is usually formulated using using (a series of) differential equations.

[vultekst om twee stukken aan elkaar te knopen]

While not used to simulate the specific functioning of computing hardware, continuous models can still play a role in contemporary research. Using a model based on mathematical equations, certain characteristics of a given hardware component can be examined. For example, Berg et al. [2006] describes a method of predicting caching behaviour of a program, in particular data locality in the context of multi-threaded applications, based on a mathematical model. A given application is executed and during this execution certain architecture-independent data is collected. Following the execution, this collected data is fed into a series of equations, each of which aims to predict certain metrics. Another metric which can be modelled using continuous simulation is the power consumption of a given architecture [Kohl et al., 2016].

From these two examples, it becomes apparent that in the context of digital system components and system architectures, a continuous model does not satisfy the need for accurate functional simulation.

For the purposes of simulating the functionality of system architectures as performed in this research, continuous simulation and continuous models have been deemed impractical, and inappropriate to the particular needs of the simulator. As will be discussed in the following subsection (2.3.2), there exists a more suitable simulation technique with respects to the functional simulation of digital system architectures.

2.3.2 Discrete-time simulation

Converse to the aforementioned continuous simulation there exists discrete simulation. Instead simulating a system with a theoretically infinite time resolution, the state of the system is simulated at discrete time intervals. A helpful analogy to illustrate the difference between these two simulation techniques might be that of the sampling of audio signals—technically any electronic signal. From a purely physical perspective sound is just a difference in pressure propagating through a gaseous medium. This phenomenon is continuous, as at any point in time the system has a certain state. When we want to record this sound however, the continuous nature poses a problem, as recording the signal with an infinite time resolution would require an infinite amount of storage. Instead, the signal is measured at a specific regular interval. From these sampled measured values, the original signal can be reconstructed. This deference between an original signal (red) and sampled values (blue) is illustrated in figure 2.1.

Discrete event modelling

While discrete modelling can be done using equations, as was true for its continuous counterpart, discrete simulation can also be based on the occurrence of events and the relationship between these

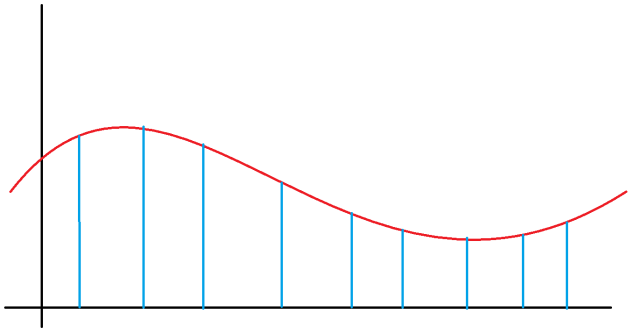

Figure 2.1: [TEMPORARY FIGURE] Continuous signal versus sampled values

events. Such event-driven modelling lends itself particularly well to the simulations of systems where there is a clear action-reaction relation between actors. As will be further illustrated in section 2.5, this latter characteristic holds true for computer system architectures, where components react to a given stimulus and this reaction might in turn be the triggering stimulus for one or more other components.

Choosing which events are allowed to be handled at a particular stage of the simulation depends of whether the simulation is either time driven or event driven. In the former case, handling of events takes place based on the the clock inside the simulation. Inside such a simulation, there is a global time and events are scheduled based on their starting time.

An example of this technique **[EXAMPLE]**

In case of the event driven approach, events are handled as they are triggered by other events. There is no set regular time between each simulation step, instead, timing between stages is based on the duration of events and the order in which new events are triggered.

this technique can be illustrated using a slightly tweaked version of the example of the former technique. **[EXAMPLE]**

2.4 SystemC

Central to this research is the C++ class library SystemC [Association et al., 2011]. SystemC is a library aimed at allowing the user to implement hardware functionality on various different levels of abstraction. [mist nog spul]

What makes SystemC especially suitable for this research is the way it allows for multiple levels of abstraction. On the one hand, using SystemC, an entire system architecture can be constructed by combining atomic logic gates who's individual implementation involves trivial bit operations. On the other side of the spectrum, SystemC allows for the implementation of a components complete functionality using C/C++ code, while still using SystemC as its interface to communicate with other components.

With the use of the example module in listing 2.4, the sections below will illustrate the basic principles and functionality of SystemC.

```
1  SC_MODULE(example)
2  {
3      sc_in<bool> clock{ "CLOCK" }, poll{ "POLL" };
4      sc_out<sc_uint<20>> out{ "VALUE" };
5      and_g and1{ "AND1" };
6      not_g not1{ "NOT1" };
7
8      sc_signal<int> sig{ "SIG" };
9
10     void handle_poll()
11     {
```

```
12          out.write(1337);
13      }
14
15      void handle_clock()
16      {
17          while (true)
18          {
19              out.write(5318008);
20              wait();
21          }
22      }
23
24      SC_CTOR(example)
25      {
26          SC_METHOD(handle_poll)
27          sensitive << poll;
28
29          SC_THREAD(handle_clock)
30          sensitive << clock.pos();
31
32          and1.out(sig);
33          not1.in(sig);
34      }
35  };
```

Listing 2.4: SystemC example module

2.4.1 SystemC components and constructs

Data types

Apart from the new classes it introduces, SystemC also comes with some data types which are useful in a hardware design context.

`sc_int<w>` and `sc_uint<w>` are the respective signed and unsigned fixed width integer types in SystemC. Using the template variable `w`, the amount of bits used to represent the value of the integer in question can be specified. These data types ensure the correct bit width independent on the architecture it is compiled on, as opposed to C/C++ built-in types like `int`. The only limitation of these types is the fact that they are limited to a width of 64 bits. In order to use integer values with arbitrary bit width, `sc_bigint<w>` and `sc_biguint<w>` can be used as the respective counterparts of the aforementioned types.

In order to streamline the passing of and the operations on groupings of individual bits, SystemC includes a so called bit vector in the form of `sc_bv<n>`. With this vector comprising of `n` bits, each individual bit can be addressed and manipulated. This eliminates the need to perform bit masks on existing data types like `int` and allows for the formation of more flexible amounts of bits—other than the usual 8, 32 and 64 bits.

When dealing with true hardware signals, value analogies using only two values—i.e. 'true' or 'false' or 1 and 0—actually don't suffice. In the real world, the values of a signal are characterised using one of the following possible categories: false (0), true (1), intermediate value (X) and floating value (Z). These possible values can be encoded in SystemC using the `sc_logic` data type. Several logic values can be combined using the `sc_lv<w>` type.

Modules

The fundamental building block of any SystemC-described hardware component is the module. Modules are packets containing the internal structure of a given component and its functionality. Listing 2.4 shows the definition of a new module called `example` (line 1-35) and the instantiation of 2 existing modules `and_g` and `not_g` in lines 5 and 6 respectively.

Ports

In order to facilitate in- and outbound communication from withing modules, SystemC offers so called ports. The three basic types of ports `sc_in<T>`, `sc_out<T>` and `sc_inout<T>` handle inbound, outbound and a combination of the two respectively. Each port has to specify the type of data which is transported by way of this port using the template variable `T`. In the example in listing 2.4, the ports of the module are declared in lines 3 and 4. In this case, there are two input ports of type `bool` and one output port of type `sc_uint<20>`. Processes can read from a port using the `read()` method and write to a port using the `write([value])` method.

Signals

The aforementioned modules with their respective ports exist on their own, but somehow need to be connected. To establish this connection the supplied type `sc_signal<T>` can be used. As with the ports, for each signal the desired data type which is to be transported along the signal is specified using template variable `T`. In the example in listing 2.4 a new signal is declared in line 8 and the signal is connected between the output of the and gate in line 32 and the input port of the not gate in line 33.

2.4.2 Processes

Having described the structural components of a SystemC hardware design, the following outlines the way in which functionality of components can be implemented in SystemC. As described in the introduction of this section, functionality of SystemC can be constructed on various levels of abstraction. The functional implementation of a component like a half-adder can be done by combining logic gates inside a overarching module, or bu implementing the addition logic inside the half-adder module using C/C++ code. The latter of these techniques and also the functionality of the logic gates used in the former technique both rely on the use of processes inside a SystemC module.

Processes inside a SystemC module come in two main distinct varieties, namely methods and threads. The difference between these two can be explained through their analogous relation with methods—or more generally functions—and threads in ordinary programming. Methods are executed each time when need be, creating a new execution context each time. Threads on the other hand run continuously. Their execution can be halted using the `wait()` function; which can optionally be given an amount of simulation time as an argument to define how long the thread should be in a non-active state. Because of the nature of this thread, remaining existent in-between execution instances, the execution context, like for example local variable definitions, are retained.

These processes are executed based on the a given sensitivity to a value change of a given component port. The syntax for this definition of sensitivity is illustrated in listing 2.4 on lines 27 and 30. In the case of line 27, method `handle_poll()` is executed each time the value on port `poll` changes. For line 30, the code specifies that thread `handle_clock` is executed each time the clock port `clock` has a positive edge. This last sensitivity definition is interesting in two ways. First of all, this is a definition pertaining to a thread, so instead of repeated execution of the function in question, a thread is started and executed until a `wait()` is encountered. Whenever the sensitivity conditions of the thread are met, the thread resumes execution until another `wait()` is executed. Second off all, this thread is not sensitive to any value change of the specified port, but only to a change which is the result of the positive edge of the incoming signal of the port in question.

Simulation scheduling

An important aspect of the SystemC simulation framework is the way in which processes are scheduled for execution by the SystemC simulation kernel. Since this research aims to simulate accurate execution of a given hardware model, it's adamant that the execution of component processes takes places in the correct order, as to result in the outcome desired by the hardware specification.

Simulations in the SystemC kernel run in a single thread. This means that any concurrency required by the model is simulated by the kernel. The processes as mentioned in the section above are called based on events. Events are the way in which SystemC handles the passage of time. These events

are triggered by certain hardware specifications, like for example the triggering of a signal, or by the passage of simulation time as described by the `sc_time` datatype, which is internally represented by an unsigned 64 bit integer.

Each time an event is triggered, the corresponding process is

2.4.3 Constructing hierarchy

SystemC allows for the embedding of components according to a desired hierarchy. This allows for the construction of more basic types, which in turn can be combined to form more complex structures. In principle, this means that based on mere atomic logic gates (e.g. AND, OR, NOT and XOR), or even single transistors, a whole system architecture can be implemented. A rudimentary example of this process is given in listing 2.4, where through the combination of an AND gate and a NOT gate inside the the `example` module using a signal, a new NAND gate is constructed.

2.5 System architectures

2.5.1 Common system architectures

2.6 VHDL

In order to translate a hardware design into actual hardware an HDL (Hardware Description Language) can be used. VHDL (the V being an abbreviation for Very High Speed Integrated Circuit) is an example of such a language, as described in the IEEE specification Lipsett et al. [1986].

The basic functionality of VHDL rests on the construct of signals

Combining the functionality

Additionally, functionality can also be implemented using sequential logic.

2.6.1 SystemC to VHDL conversion

Of interest to this project, is the ability to convert SystemC hardware descriptions to a VHDL-based implementation. On this topic, some research has been conducted already. The first of these earlier works is Côté and Zilic [2002].

More recent work on the conversion of SystemC to VHDL is outlined in Chen [2011].

CHAPTER 3

Implementation

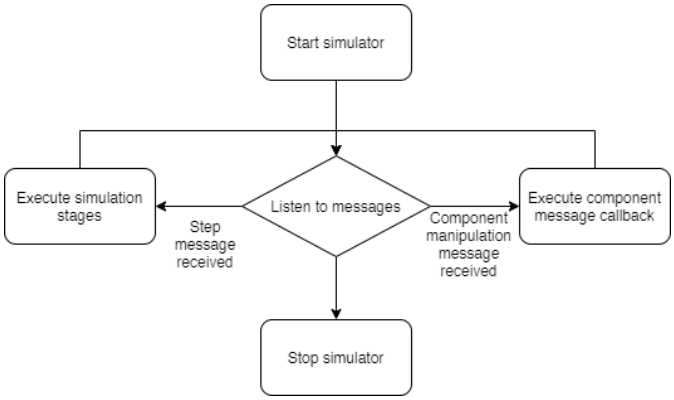

Figure 3.1: Flowchart of simulation process execution

This section presents an outline of the implementation details of this research. At first, the specific choices made when it comes to the front-end used to control and monitor the simulated architecture will be illustrated.

Another important part of the implementation outline are the specifics dealing with the creation of compatible SystemC architecture implementations is discussed. The framework, as implemented in this research, offers a set of tools in order to simplify the creation process.

Additionally, the specific implementation of the architectures used in order to perform the experiments, whose results are shown in chapter 5, are detailed.

3.1 Architecture simulation

3.1.1 Process communication

For this framework, a choice had to be made on the way the simulation front-end and the SystemC-based hardware model would need to communicate. Given the nature of SystemC, it being a library for a compiled programming language, any changes in the hardware specification would result in recompilation of (specific parts of) the C++ code. Were the

Another reason for choosing this layout when it comes to the relation between the framework front-end and any architecture model is the

Finally, since

With this approach of communications also come some drawbacks. The first of which is added

The second possible problem which could be introduced by this communication method

Communication channels

Messages

```
{
    "message_type"     : string,
    "message_sub_type" : string,
    "destination"      : string,
    "origin"           : string,
    "data"             : ...
}
```

Listing 3.1: Main message lay-out

API
In addition to the

3.1.2 Simulation front-end

Component visualisation and control

3.2 SystemC synthesis

To perform experiments with a given computer architecture, it will first have to be implemented using SystemC. As a library, SystemC offers a lot of flexibility when it comes to implementation, and, as such, leaves many choices up to the user. However, for the use of an SystemC-implemented architecture in this simulation framework, some details of the design choices need to be make beforehand. This allows for a more seamless integration between the simulated architecture, the framework front-end and external applications connected to the front-end using the API detailed in section 3.1.1.

3.2.1 Basic components

In order to aid the creation process, a number of basic components, often present in system architectures as described in section 2.5. These components serve as the basis of the construction toolkit, which can be expanded using any number of custom implemented SystemC modules. By offering a set of basic components with certain predetermined characteristics, integration of the architecture in the existing communication and front-end infrastructure—further expanded in sections 3.1.1 and 3.1.2 respectively—is made seamless. This integration is done by offering preexisting handles for sending event messages The basic structure and functionality of these components will be outlined using a short description of its basic functionality.

Program counter
While being a comparatively simple component when compared to the parts discussed hereafter, the program counter plays an important role in the monitoring and control of a running simulation. The module offers a clock input port, which receives a signal upon which the counter is incremented with a given incremental value, this

Register file

Memory

Basic arithmetic component
Lastly, the basis for arithmetic units is provided. This module can either take an optional clock signal as its functional initiation, or can act as a more passive component and use its data input ports as triggers for its logic. The arithmetic component can perform a single pre-programmed arithmetic operation, or a dynamically selectable range of operations. In case of the former, a passive implementation without clock signal is appropriate. In the latter case however, it is useful to implement functionality using the clock signal, for this enables the sending of an event with each instance of the unit performing an operation. This event can broadcast the operation performed, and each of the operands involved.

Instruction	*Op-code*	*Arguments*
add	0x0	*destination register, oprand register 1, oprand register 2*
addi	0x1	*destination register, oprand register 1, immediate value*
jz	0x2	*jump address*
mv	0x4	*destination register, source register*
	0x5	*destination memory address, source register*
	0x6	*destination register, source memory address*
	0x1	*destination memory address, source memory address*

Table 3.1: Instructionset of TA4

The arrangement of data input and output ports and possible control and flag signals it left to the user to implement, as this could have any number of configurations depending of the functionality of the component.

3.2.2 Event staging

[intro]

The following is an example of a sequence of stages, as used by the testing architecture as discussed in section 3.3.1.

1. Instruction fetch
2. Control set
3. Data read
4. Data write
5. Program counter update

Internal clock

In order implement the staged cycle, an extra component is added to each architecture written to be used in this framework.

3.2.3 Architecture configuration

3.3 Testing architectures

For testing purposes,

3.3.1 TA4-SC

Instruction-set architecture

Before implementation of the testing architecture could commence, an ISA (instruction-set architecture) had to be made. Since this architecture would primarily be used for testing purposes, the ISA would not have to be as extensive as those for architectures like x86, ARM or MIPS. Therefore, only a few instructions were selected for implementation. The selection was done carefully however, to ensure that while the architecture would be basic, its limited instruction-set could theoretically be used to implement somewhat sophisticated assembly programmes. The chosen instructions and their argument layout is shown in table 3.1.

The ISA features two arithmetic instructions, namely `add` and `addi` (add immediate). The first (`add`) stores the result of an addition of the data inside two given operand registers into a given destination register. The latter (`addi`) also stores the result of an addition into a given destination register, however, its operands are the data from a given operand register and an immediate value encoded inside the instruction byte code.

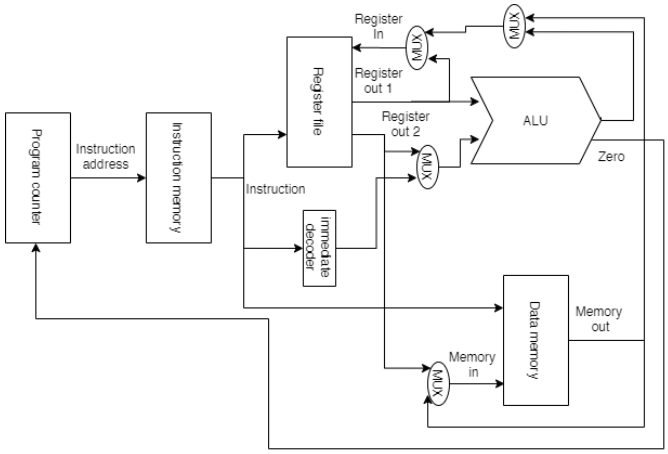

Figure 3.2: Schematic diagram of the data paths in the TA4-SC architecture

To facilitate flow control, the instruction set has a conditional jump in the form of a `jz` (jump-zero) instruction. As the name suggests, upon execution of this instruction, a jump to a given instruction address takes place if the zero flag of the ALU is set to true. One might wonder why the choice was made to handle all flow control using only one instruction, instead of using a separate jump and branch instruction. As mentioned in the introduction, the aim of this ISA was to keep it simple, while maintaining the support for more advanced programmes. By preceding a jump zero with an instruction which is sure to result in the zero flag being set to true, the combination of these two instruction turns into an ordinary jump instruction.

Data movement between RAM and registers or between two locations inside these two components is handled using the `mv` (move) instruction. Although it may seem like this is one single instruction, the different opcodes—as featured in table 3.1—are inferred during assembling using the given instruction arguments. The instruction takes two arguments, the first being the location to which the data is to be moved, and the second being the origin of the data. Either argument can be the name of a register or an address in RAM. Based on this type, one of four opcodes is used.

Listing 3.2 features an example program using the ISA4 instruction set.

```
1 ADD  $0, $0, $0
2 ADDI $1, $0, 3
3 MV   1,  $1
4 MV   $2, 1
5 ADD  $0, $0, $0
6 JZ   5
```

Listing 3.2: TA4 instruction set assembly example program

This particular ISA features a few extra characteristics which are not immediately evident from the instruction listing in table 3.1. First of all, the ISA dictates that the contents of register `$0` always remains 0. This way, and instruction whose byte-code value is equal to zero—which incidentally is equal to the first line of assembly in listing 3.2—can be seen as an implicit `nop` (no operation) instruction, since no data would be manipulated. The second extra feature present in this ISA is that, if a jump-zero were to take place to the current program counter address, the instruction execution halts. With this feature, a program implemented using ISA4 is able to stop in a controlled manner.

The lay-out of these instructions dictate some of the characteristics of the hardware components which need to be constructed to use this ISA. For example

Architecture implementation

Based on the ISA, an architecture was designed which supported each of the outlined instructions. This first implementation design is illustrated schematically in figure 3.2.

3.3.2 TA4-PL

SystemC implementation

CHAPTER 4

Experiments

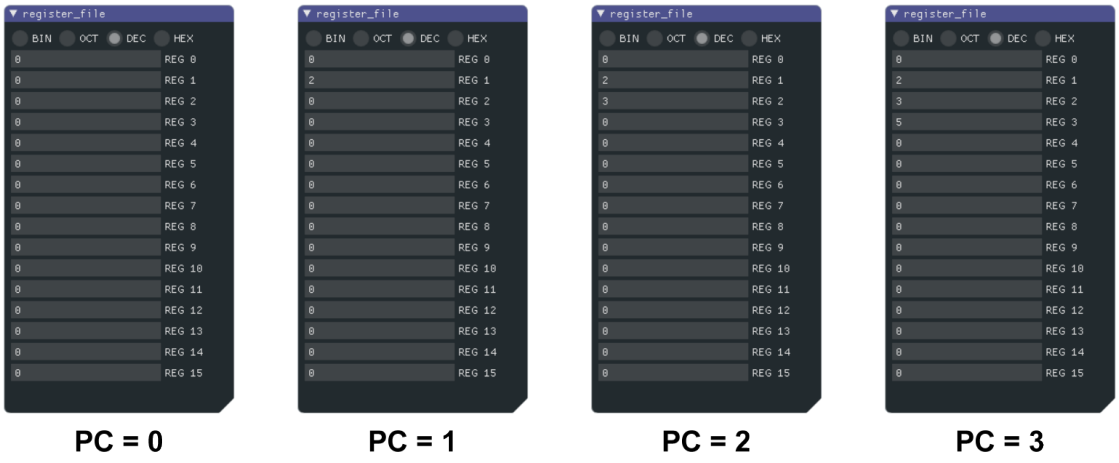

Figure 4.1: Contents of the register file during the execution of example program **??** for each time step

4.1 Computational correctness

4.1.1 Test program: simple addition

```
1 ADDI $1, $0, 2
2 ADDI $2, $0, 3
3 ADD  $3, $1, $2
4 MV   0,  $3
```

Listing 4.1: ISA4 assembly simple addition and memory move

4.2 Simulation framework overhead

4.2.1 GUI-simulation communication

4.2.2 Component manipulation

4.2.3 Program execution

4.2.4 Message integrity

4.2.5 External interactions

4.3 Extensibility

4.3.1 Component manipulation

4.3.2 Execution correctness checking

CHAPTER 5

Conclusion

Appendices

APPENDIX A

Continuous simulation in detail

A classic example of a system which can be simulated using differential equations is the size of a given population [Birta and Arbez, 2013]. Let's say there is an animal population with a size $P(t)$ where t is a certain point in time. This simple definition already highlights the continuous nature of such a model, as the equation simulates the population size for a continuous time domain. With the population size as stated above, the population increase at a given point in time can be formulated using the differential equation $\frac{dP}{dt} = b(t) - d(t)$. In this equation $b(t)$ equates the rate of birth and $d(t)$ the rate of death at time t. Using the following system of equations, this model can be elaborated further into a full fledged model:

$$b(t) = P(t) \cdot 0.5 \tag{A.1}$$

$$d(t) = P(t)^2 \cdot 0.03 \tag{A.2}$$

Parameters like 0.5 and 0.03 as used in the example above, are determined based on experimentation or observations as they occur in the system which is sought to be simulated or based on assumptions about this particular system. Having formulated these equations, their behaviour can be plotted using a vector field. With such a diagram, expected behaviour will become apparent. This is also the case for the example model, as can be seen in its corresponding vector field in figure A.1, where, for example, a clear equilibrium point can be observed at $P(t) \approx 26.5$. Based on these comparatively simple operations, one is able to predict that, in the case of this example population, any population with an initial size of $P(0) < 26.5$ is expected to increase in size as time passes, whereas a population with an initial size of $P(0) > 26.5$ will shrink according to this model.

Although the example given above is rather simplistic, continuous models allow for the simulation of systems with more complexity. As long as equations can be devised which accurately describe the desired characteristics of a system, such a system should theoretically be modelable using continuous methods. Herein lies however also one of the disadvantages of continuous models, as the success of a given model greatly depends on the ability to formulate the appropriate equations. This fact is describes in Kohl et al. [2016] as it applies to the modelling of hardware using equations.

Application in hardware modelling

Knowing this, one might wonder how such techniques could be applied to the simulation of computing hardware, as these components would on a first glace not lend themselves to a behavioural description using mathematical equations. As it tuns out however, behaviour of fundamental electronic components like capacitors, resistors, inductors and voltage sources and the the way these parts interact with each other inside a circuit can also be modelled using mathematical equations. This fact might not be too surprising to anyone with an above-average background in the field of physics as it pertains to electricity.

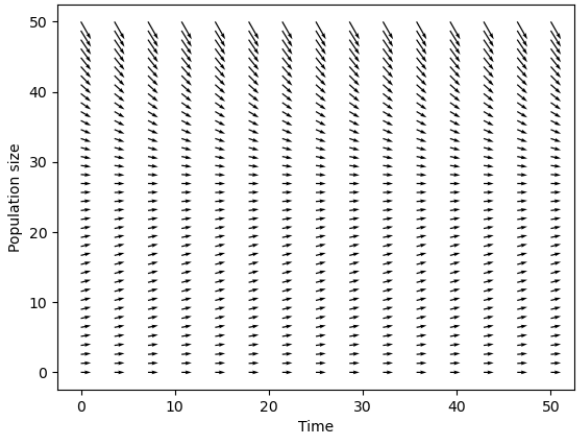

Figure A.1: Vector plot of the differential equation system as described in section 2.3.1

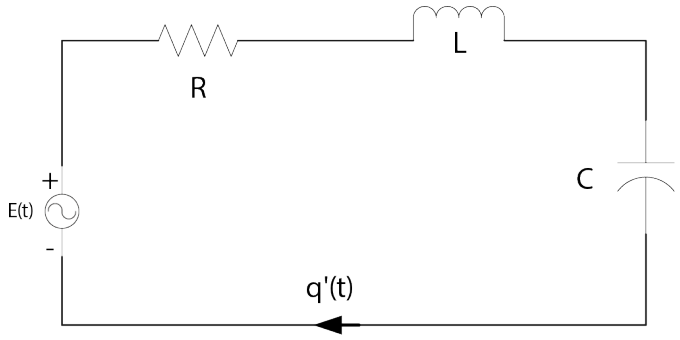

Figure A.2: Example circuit [Birta and Arbez, 2013, p. 251]

An example of this ability to model electrical circuits using continuous modelling is featured in Birta and Arbez [2013]. A circuit, as features in figure A.2, contains a voltage source ($E(t)$), a resistor (R), a capacitor (C) and an inductor (I) in series. The book formulates an equation (equation A.3) which is able to model the state of the circuit based on Kirchoff's voltage law. Without getting into extraneous detail about the specifics of electrical engineering, it follows from this equation that the state of the components present inside the circuit can be derived from the initial values of the state variables ($q(t)$ and $q'(t)$ at t_0) and the value of the input variable ($E(t)$).

$$Lq''(t) + Rq'(t) + \frac{q(t)}{C} = E(t) \tag{A.3}$$

These general characteristics of electronic components have also been used in a more computational context. In the past, the use of electrical analogue computing equipment, like the machine in figure A.3 were used to perform simulations on physics models—e.g. tidal calculations or certain models used by the oil industry. These analogue computers contain circuits with basic electrical components whose electrical properties are similar to certain mathematical constructs. As such, a circuit built from the appropriate components is able to be used as the electrical analogue of an equation. When looking at the simulation of such machines, we are presented with a peculiar chicken and egg problem. It is clear that the analogue machine in this example was used for continuous simulation purposes, and as such, simulating it using a continuous model should logically be possible. While no example of such a simulator appears to exist, is need be, it should be possible to device a model to simulate the functionality of such a computer.

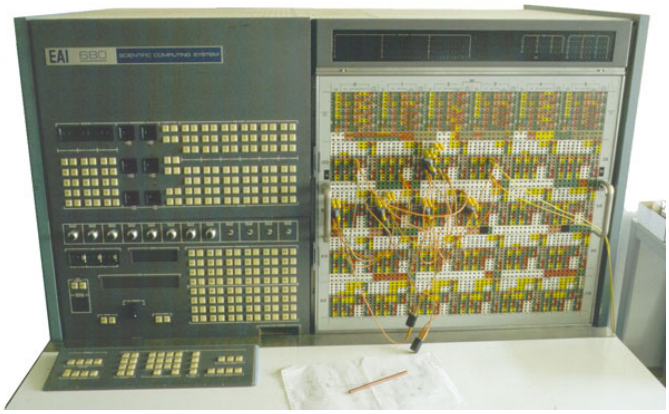

Figure A.3: Electronic Associates Inc. Model 680 [Dooijes]

Bibliography

IEEE Standards Association et al. Ieee std. 1666–2011, open systemc language reference manual, 2011.

Erik Berg, Håkan Zeffer, and Erik Hagersten. A statistical multiprocessor cache model. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 89–99. IEEE, 2006.

Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

Louis G Birta and Gilbert Arbez. *Modelling and simulation*. Springer, 2013.

Rui Chen. Synthesizable systemc to vhdl compiler design. 2011.

Christian Côté and Zeljko Zilic. Automated systemc to vhdl translation in hardware/software codesign. In *9th International Conference on Electronics, Circuits and Systems*, volume 2, pages 717–720. IEEE, 2002.

E.H. Dooijes. Electronic associates inc. model 680 scientific computing system. URL `https://ub.fnwi.uva.nl/computermuseum/eai.html`.

C Eibl, Carsten Albrecht, and Rainer Hagenau. gsysc: A graphical front end for systemc. In *European Conference on Modelling and Simulation*, pages 257–262, 2005.

Johannes Kohl, Marc Reichenbach, Carsten Demel, and Dietmar Fey. A systemc based framework for cycle accurate processor simulation and parameter analysis. *IFAC-PapersOnLine*, 49(25):92–97, 2016.

Roger Lipsett, Erich Marschner, and Moe Shahdad. Vhdl-the language. *IEEE Design & Test of Computers*, 3(2):28–41, 1986.

Christian Menard, Jeronimo Castrillon, Matthias Jung, and Norbert Wehn. System simulation with gem5 and systemc: The keystone for full interoperability. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 62–69. IEEE, 2017.

Sandro Rigo, Guido Araujo, Marcus Bartholomeu, and Rodolfo Azevedo. Archc: A systemc-based architecture description language. In *16th Symposium on Computer Architecture and High Performance Computing*, pages 66–73. IEEE, 2004.

Matt T Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 23–34. IEEE, 2007.